Simplifying A/B Testing Implementation and Management in Vaadin Application

Quang Tien Nguyen

Thesis submitted to Department of Information Technologies
Åbo Akademi University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Software Engineering

Supervised by
Professor Ivan Porres

March, 2019
Turku, Finland

ABSTRACT

Today, web technologies are developing rapidly, which offers all companies tremendous opportunities for developing web applications to solve their specific problems. While a good web application creates more chances to grow the business, it will never be perfect. There should be a strategy in place to improve it over time in order to obtain higher user engagement, which in turn could potentially result in new customers. Therefore, companies often perform online controlled experiments to offer better experiences with their applications. In other words, the operations can be called A/B tests with the web application randomly assigning a variant for a factor to random users. The users' behaviors toward the application with a specific modification is then collected and evaluated. In the end, the best variant for each factor is made available to all users so as to leverage its advantages. The approach can be applied to most of web development frameworks including Vaadin, an open-source platform which helps Java developers create modern web applications. This thesis implements a solution as a Java library, which eases the process of creating and managing A/B tests in Vaadin applications. Moreover, the library will be used in an example application to demonstrate how conveniently developers can create A/B tests in a Vaadin application.

Keywords: Vaadin, A/B tests, Vaadin Flow A/B tests, Vaadin Framework A/B tests, A/B tests in Java web development.

Contents

List of Figures

List of Snippets

Acknowledgments

I would like to express my most profound gratitude to my thesis supervisor, Professor Ivan Porres, who helped me to choose the thesis topic and offered me extensive guidance while I was conducting the research. In addition to the research methodologies and techniques, he also provided his insights on the topic, which was a source of great motivation. Without his valuable and great advice, this thesis would not have been possible.

In addition, I would like to express my deepest thanks to my brilliant colleagues in Vaadin. I have received a great deal of help and encouragement from them. Moreover, they have built a great platform which is an essential component of my thesis.

Last but not least, I cannot find the words to describe my gratitude to my beloved wife, my family, and friends who continuously support and encourage me throughout my life.

Chapter 1

Introduction

During the past decades, software technologies have attracted much attention and been developed in many ways. In particular, web applications and their related technologies have become widely popular [1]. Most companies and organizations today have a website to either provide and broadcast information or operate their business on a web platform. Regardless of the web application's purposes, owners always want users to have pleasant experiences and for the sessions to last as long as possible. Users of their web application might become customers, which could ultimately lead to an increase in the company's revenue. Therefore, companies keep improving their web applications in order to provide the best experiences and services for customers and this includes giant technology companies such as Amazon and Microsoft. They have proactively tested different application features by using online controlled experiments, or A/B testing (section 2.3). Following these trials, they have gained knowledge about the behaviors of users toward their products [2]. As a result, they have identified obstacles and made improvements to the applications.

1.1 Purpose of the thesis

In the extensive market of technology stacks for web application development, the Vaadin Platform provides exclusive tools to assist Java developers in building modern-looking web applications. Developers who use the Vaadin Platform are obviously able to do experiments with their application. However, there is no official way of supporting experimentation with a Vaadin application. The developers might need to implement the solution from scratch, which will require many trials. In this study, a generic solution for experimenting in a Vaadin application will be introduced and implemented. The primary goal is to extract the common logic for creating A/B tests into a helper library, called *ABHelper*. It should focus on simplifying the process of experimentation in Vaadin and provide reusability, so that everyone can benefit from and create their own A/B tests with fewer efforts.

1.2 Thesis structure

The thesis will provide the implementation details for *ABHelper* with the following structure:

Chapter 2: Background The background chapter will focus on providing information and definitions for related technologies and terms which are used in this study.

Chapter 3: Project's implementation This chapter will reveal the architect of the *ABHelper* library and its details.

Chapter 4: Demonstration An example application will be applied to the *ABHelper* to evaluate which points it has improved the process of creating A/B tests in the Vaadin application.

Chapter 5: Advantages and limitations A summary of the project's evaluation will be provided in this chapter.

Chapter 6: Conclusion and future work This chapter provides a conclusion about the work which has been done and the results achieved in the research.

Chapter 2

Background

2.1 Web application

A web application is a specific type of software application which is hosted online and accessible via web browsers. Instead of showing static information like traditional websites, it solves particular problems by providing services for the tasks requested by ordinary users[3]. For instance, a web application could provide email services which help users send or check their emails via the web browsers[1]. It can also find the cheapest flight tickets[2] or provide online collaborative tools[3].

With the extraordinary growth of Internet users (1066% in the 2000-2018 period[4]), giant companies have invested incalculable efforts into improving their web applications so as to attract more users and eventually bring in greater profits. Explicitly, this motivates companies to develop optimized web applications in order to increase their user base. In particular, they constantly improve their products by fixing bugs, adding new features, designing better user interfaces (UI) and increasing application performance. The UI

[1]Gmail(https://gmail.com) or Outlook(https://outlook.live.com), visited on 17.02.2019

[2]Sky Scanner (https://www.skyscanner.fi) or Momondo (https://www.momondo.fi), visited on 17.02.2019

[3]Google docs (https://docs.google.com) or Figma (https://www.figma.com/), visited on 17.02.2019

Figure 2.1: Gmail user interface in 2004, a screenshot created by its designer, Kevin Fox[5].

comparison between 2004 and 2019 for Gmail is shown in figure 2.1 and 2.2. Ultimately, such improvements mainly share only one common goal, which is to create better user experiences (UX) in order to keep and attract users.

2.2 User experience in web application

In general, user experience (UX) refers to a user's experience with software, which in this case involves a web application. It includes the users' feelings and attitudes regarding a specific product, system or service. It is very similar to the experience when listening to

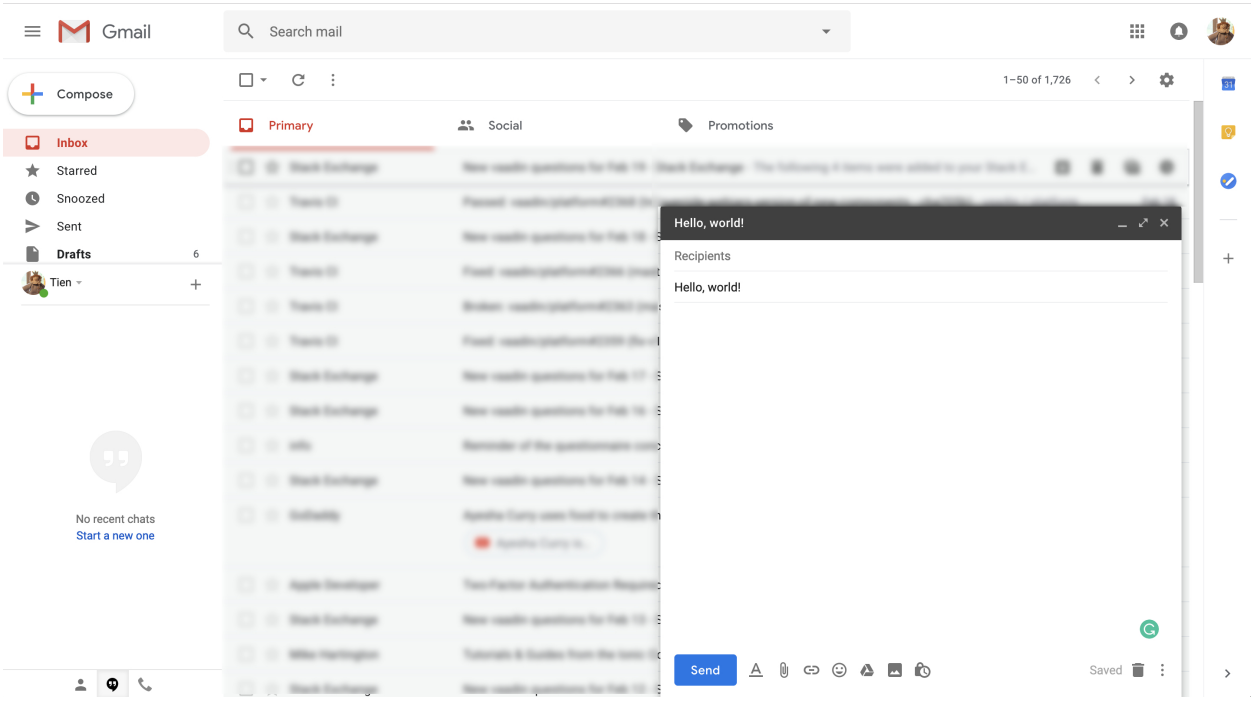

Figure 2.2: Gmail user interface in 2019.

music, insofar as software UX is subjective. A person might love jazz music and never be impressed by a pop song. This is the same for user experiences: they are subjectively favorable for some users but simultaneously negative for others [6, page 5].

There are quite a number of use cases proving the relationship between an application's UX and a company's revenue. In case of the Doctor FootCare checkout page (figure 2.3), there was a 90% loss in revenues following an upgrade from variant A to variant B. However, the company was able to recover and increase 6.5% conversion rate relative to the original version by removing the coupon field. The reasoning behind this decision was that the discount coupon field can make people wonder if they are paying a higher price since they do not have the discount code. Thus, upgrading to variant B without the coupon field led to a better UX and implicitly boosted their business [2]. In other cases, a better UX can also result from improvements to back-end algorithms. However, companies used the same method to achieve their conclusions by performing controlled experiments which are also known as A/B tests.

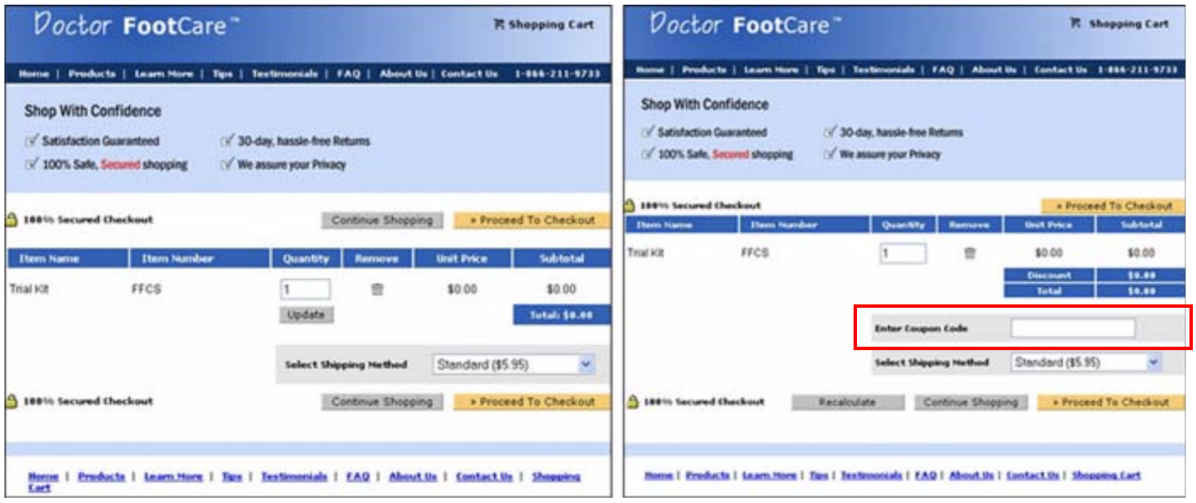

Figure 2.3: Doctor FootCare's checkout page. Variant A is to the left and variant B to the right.[2, Figure 1]

2.3 A/B testing

It is the fact that contents of web applications can be quickly updated and enhanced. This creates a short cycle of improving the content, delivering it and collecting feedback based on the changes. This makes it possible for developers to quickly try different approaches for a targeted objective. Subsequently, the collected data and feedback can be analyzed to form a better idea for the tested units. This is the basic idea of A/B tests, which are also called online controlled experiments, split tests, Control/Treatment tests or parallel flights[2].

In the basic setup of an A/B test, users of the application will randomly receive one out of two (or more) variants. The first variant is often the default one which is currently in use, while other variants are options that the developers evaluate for potential use instead of the default one. Based on the data collected from users who get different variants, people will make decisions to choose one of the new options or keep the default one. Frequently, a winning variant should have a significantly better impact compared to the others.

In practice, such experiments can have unexpected results. In 2006, Greg Linden at Amazon created a prototype to show personalized recommendations based on the items customers have in their cart[7]. He was forbidden to continue the work on this feature because the marketing department thought that the customers might be distracted from the checkout process and ultimately abandon their cart. It was a valid concern, although Linden did not believe it and rolled out an online experiment for the feature. The statistics showed that he was right and Amazon decided to launch the feature officially for all Amazon customers.

2.3.1 Terminology

There are some terms broadly used in literature related to A/B tests and experiments. They are also repeated throughout in this study.

Overall Evaluation Criterion (OEC)[8]: This term refers to the acceptance condition of an experiment which is usually expressed as a number for a measurable metric. However, an experiment may have more than one objective, and in such cases it is highly recommended to formulate a single index (called OEC) from all the criteria [8, page 52].

Factor: This is a manageable variable in an experiment. It might have direct or indirect effects on the OEC. When running an experiment, different values are assigned to factors. For instance, in a classic A/B test, there should be a factor with two values: A and B.

Variant: Different values of a factor are called variants. In the example above, A and B are variants of the factor. The original value prior to the A/B test is called the *Control*, while the new variant is the *Treatment*. For example, for a bug with A/B implementation, the *Control* will be the fallback value of the factor.

Null hypothesis: A hypothesis is called a "null hypothesis" when its OECs of different variants are not significantly diverse. If there is any inconstancy between the figures, it is caused by randomization without any meaningful reason.

Confidence level: This is the probability of preserving a null hypothesis when it is true. In other words, it is the ability to detect differences when they exist.

Power: The probability of rejecting a false null hypothesis.

Standard deviation (Std-Dev or σ): This is a measure which is used to quantify the variation of a value set.

Standard error (Std-Err): In statistics, standard error is the standard deviation of its sampling distribution. The relation of Std-Err and the sample size is: Std-Err=$\sigma/\sqrt{n}$ where σ is the estimated deviation and n is the sample size[2, page 151].

2.3.2 Defining an A/B experiment

Conducting an A/B experiment will help to understand and validate a hypothesis in the target application. For instance, for an e-commerce website, only 10% of the customers actually start the process of checking out after adding items to the shopping cart. A typical assumption could be that the procedure is counter-intuitive or not readily perceptible to customers. Thus, to increase the percentage of checkouts, a hypothesis could be established based on making the checkout button more visible by changing its color from blue to green. Therefore, the checkout button color is a factor in this experiment. The factor has two different variants: green and blue. The OEC for this experiment would be the percentage of customers starting the checkout process.

From the above example, the main steps for defining an experiment can be listed as:

1. Identify the problem by analyzing the conversion flow and determining the obstacles.

2. Formulate a hypothesis. This is an important phase, since it will affect the decision to choose the factors and OCE. The hypothesis could be the best prediction as to what the obstacles might be.

3. Define testing factors and their values based on the determined hypothesis. The factors are components in the application that could be executed differently. It might be a visible feature, back-end algorithms or user flow.

4. Choose an OEC for the factors. With a statistical difference in the OCE, it should be possible to easily conclude that the chosen hypothesis is true and that it is logical to apply the winning variant to the application.

2.3.3 Evaluating experiment result

After running experiments, the collected data needs to be analyzed and evaluated to determine if the *treatment* should replace the *control*. There are two useful formulas [2, page 152] for running and validating the results.

The first formula (2.1) is used to calculate the minimum sample size that could provide a reliable conclusion. It also helps to measure the testing time depending on the amount of users visiting on the target application.

$$n = \frac{16\sigma^2}{\triangle^2} \tag{2.1}$$

In formula 2.1, it has been assumed that the *confidence level* is 95% and the *power* is 80%. n is the number of users for each of the variants. If there are multiple factors in the experiment, it is assumed that the number of variants for each factor is the same. σ^2 is the variance of the OEC and $\triangle$ is the difference to be detected. The coefficient of 16 in the formula corresponds to *power* 80%. In the event that an increase of the power to 90% is desired, the coefficient should become 21, which means there is a 90% probability of

rejecting the *null hypothesis* if there is no difference between variants. With the number of users for each variant, you can estimate the time needed to run the experiment based on the application traffic.

Formula 2.2 determines if the formulated hypothesis is correct or not. $\overline{O_A}$ and $\overline{O_B}$ are the average of the OEC of variant A and B. $\widehat{\sigma_d}$ is the estimated standard deviation of the difference between OECs. Based on the *confidence level*, a threshold *t* is formed: for example, with a 95% confidence level, and the threshold can be set to 1.96 [2]. If the absolute value of the threshold *t* produced from the formula 2.2 is greater than the defined value, it means that there is a statistical difference between the *treatment* and the *control*. As a result, the *control* should be replaced by the *treatment*.

$$t = \frac{\overline{O_B} - \overline{O_A}}{\widehat{\sigma_d}} \tag{2.2}$$

2.4 Vaadin Platform

The Vaadin Platform[4], formerly the Vaadin Framework, is an open-source project created and maintained by Vaadin Ltd. The platform offers a web development ecosystem which involves multiple products which are listed as:

Flow is the core framework for connecting and binding client-side Vaadin components to the server side. It provides pure Java application programming interfaces (APIs) to create and interact with client-side HTML and JavaScript.

Components comprises a set of high-quality web components which could be integrated into Vaadin Flow or used independently in any modern web applications.

Tools are extra tools which can help developers improve productivity and produce better web applications. It includes Vaadin Designer (a plugin for Eclipse[5] and IntelliJ

[4] `https://vaadin.com` (visited on 27.03.2019)

[5] Eclipse is an IDE for various programming languages: `https://eclipse.org` (visited on 27.03.2019)

IDEA[6]), Vaadin Testbench (a library to write regression tests for Vaadin Components in Java) and Multiplatform runtime (a migration tool for migrating from a deprecated Vaadin version to the new one). Moreover, Vaadin also has various base projects with different purposes to help people get started quickly. The example application in chapter 4 is one of these.

The Vaadin Platform is the choice for this research because it offers unique features in the current market. The core frameworks are open-source, so it is useful to investigate and understand how the code works. Moreover, Vaadin provides great APIs, including opportunities to attach the solution from this research as an add-on.

Currently, Vaadin is running on a release train model which has two different version types: regular release and long-term support LTS release. During the time this research was conducted, a few new versions were released. The solution that is currently being used is the latest regular version which is 13.0.2 [7].

[6]IntelliJ IDEA is a popular Java IDE: `https://www.jetbrains.com/idea/` (visited on 26.03.2019)
[7]`https://github.com/vaadin/platform/tree/13.0.2` (visited on 27.03.2019)

Chapter 3

Project's Implementation

As a solution for creating and managing A/B tests with the Vaadin Platform, this project has been developed as a Java library which uses Maven [1], a prominent way of managing dependencies and build processes for Java projects. The implementation is stored in a GitHub repository at https://github.com/qtdzz/ab-helper.

3.1 Analysis

3.1.1 Overall workflows

This project focuses on two main types of AB testing techniques: non-redirecting (including A/B/n and multivariate tests) and redirecting mechanisms. They share some common management classes such as the static controller which manages and applies different variants on a component based on the defined strategies. The non-redirecting technique is illustrated in figure 3.1 while the URL redirecting approach is illustrated in figure 3.2. *ABController.java* will mainly provide the public API for variant assignment with the help

[1] https://maven.apache.org/what-is-maven.html (accessed on 11.11.2018)

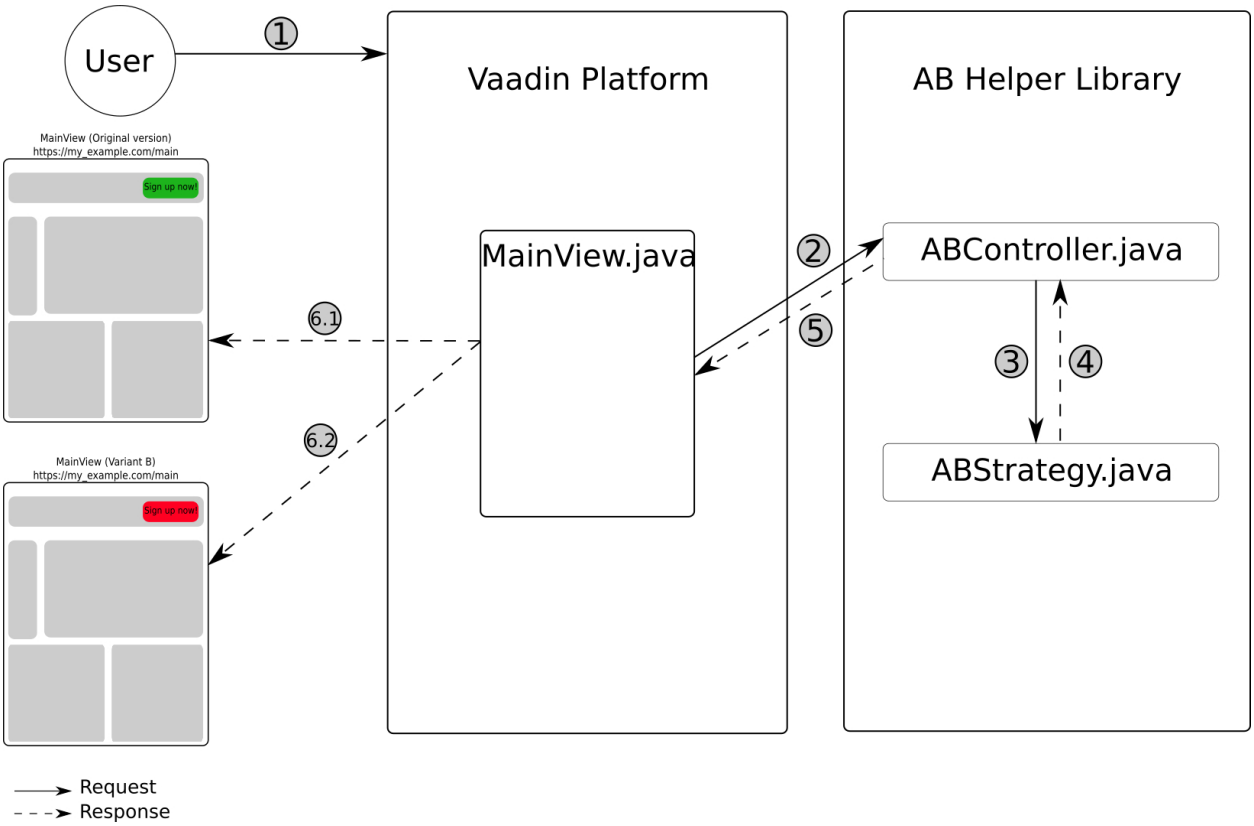

Figure 3.1: Non-redirecting AB testing workflow.

of *ABStrategy.java* where the randomization is executed to choose a suitable variant.

In a non-redirecting approach, when receiving requests from a user, the AB helper library is used to assist in the determination of a variant for the current user. For the redirecting path, the AB helper library will determine whether to stay on the current URL or redirect to the alternative ones which contain variants of the experiment. For both approaches, the decision is saved for each user to ensure that the user will not be confused because variants keep changing [2, Randomization algorithm, page 163].

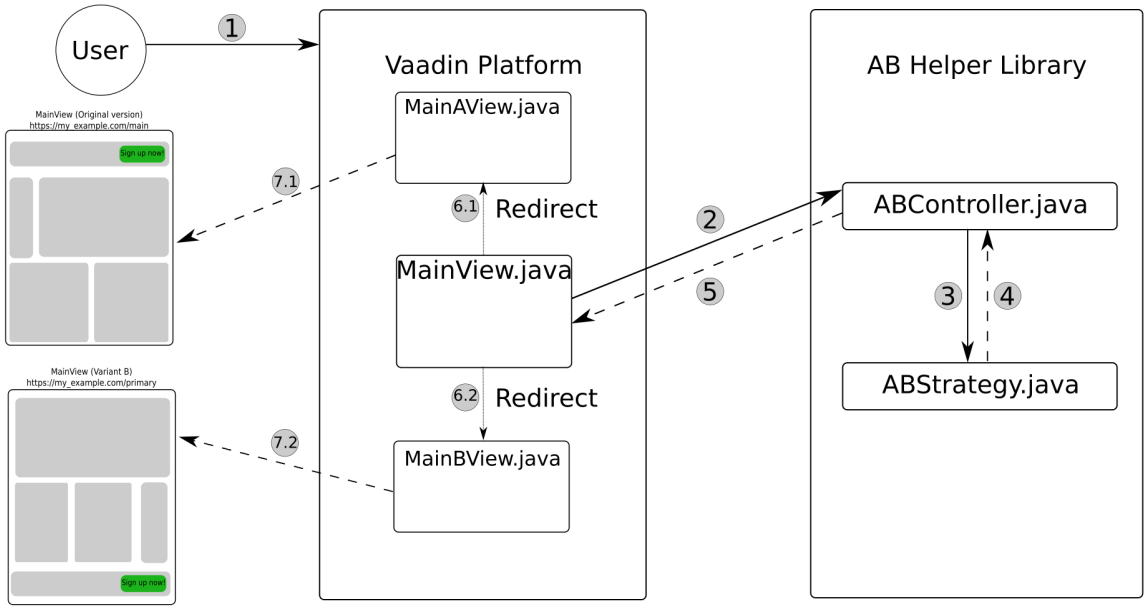

Figure 3.2: Redirecting AB testing workflow.

3.1.2 Use cases

Use case 1: style variants

The *ABHelper* library should help developers test various style variants on components. In the Vaadin context, this means that developers are also capable of adding different styles, themes, and visibility options. This feature enables developers to implement different visual styles of components by only specifying the variants they want to test.

In a case study made by **Home24** [2], one of the largest furniture stores in Europe, it was found that customers are more engaged with a home page that has a neat header with light colors and simplified menu. As a result of the test, the winning version of the home page increased the number of orders by 9% for each session and 5% more in searching usage. The variants comparison is shown in figure 3.3.

[2]https://www.home24.com (visited on 31.01.2019)

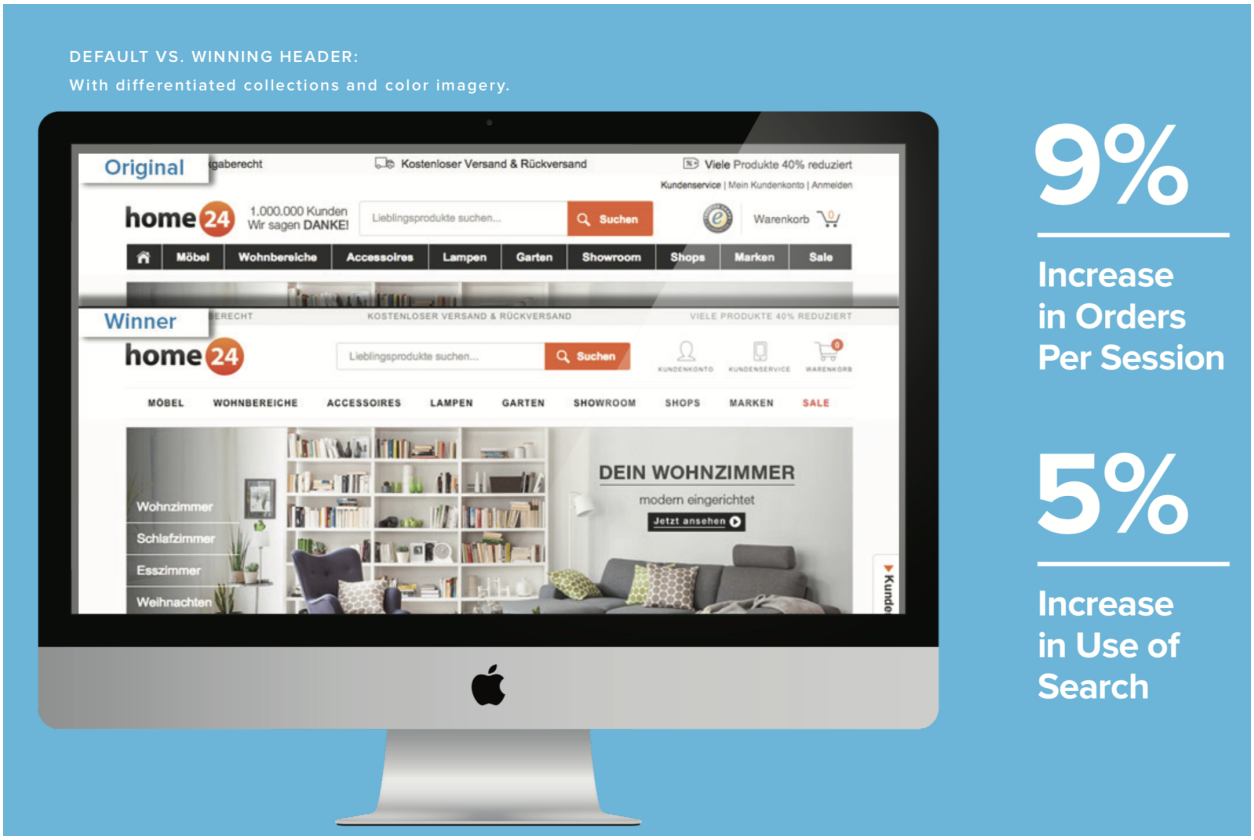

Figure 3.3: The original version compared to the winning one in **Home24**'s optimization. Source: [9, page 24]

Figure 3.4: The original text, "***Why Use Us***", compared with the winning text, "***How It Works***". Source: [10, Figure 5.3]

.

Use case 2: text content variants

Developers using the library might want to test multiple text content variants for components. With this feature, it is possible to check the effects of distinct text content or call-to-action texts for their buttons.

In an experiment by **FormStack**, a ***Why Use Us*** button on the homepage was tested using different variants. It turns out that the best call-to-action text is ***How It Works*** (figure 3.4). The winning variant brought 50% additional traffic to that page and also increased the number of two-week free-trial sign-ups to 8% [10, section "Why versus How: Formstack", page 64].

Figure 3.5: The original form compared with the winning variation which has pre-filled text in message field. Source: [9, Page 31]

.

Use case 3: value variants

For developers using the library, it would be convenient to be able to pre-fill the components with different contents. The targeted object can be a message text field in a contact form or a color select-box in an order form.

In an experiment by **AutoScout24**[3], the biggest online market for trading new and used cars in Europe, results show that the variation with pre-filled data (figure 3.5) in a contact form dominates other variations in increasing the contact conversion by 22%[9, Page 30].

[3]https://www.autoscout24.com/ (accessed on 01.02.2019)

Use case 4: view variants without redirecting URL

There is a need for testing back-end logic in the same interfaces but with different content[2, page, 127]. A salient example of this case is when developers want to test two algorithms which suggest other products while customers are browsing a specific product's pages. Therefore, two product views are used in this example; each of them should implement a different suggestion algorithm. In general, the UIs are identical; therefore it is not necessary to change the URL.

Use case 5: view variants with redirecting URL

When refactoring a view, developers want to split the users into two view variants, the new and the old, so that they can evaluate how the new view affects the conversion rate. If there is a significant decrement in the new version, they will have to address all related matters prior to deploying it as the main view. Because there might be considerable differences between two variants, it is suggested to keep the URL different for each of them so that users may revisit the same page.

Use case 6: collecting data

When using this project as a library, developers should be capable of combining experiments' data with their usual application's data. Accordingly, they can evaluate the tests' results and optimize their application to better achieve their various goals.

It is quite obvious that experiment data is an essential detail that determines the outcomes and conclusions for the experiment after being thoroughly analyzed [2, section 5].

Use case 7: managing factor

It would be beneficial for developers to be able to control the experiments without changing or removing the source code. Of course, they would need to do this when the tests are entirely irrelevant. Due to the nature of the library, it cannot provide a full managing view, but the view would be an abstract class which the developers need to implement as well as protect using their authorization mechanism.

3.2 Implementation details

3.2.1 Factor models

According to the analysis in section 3.1, the factor model can be classified into two different types. The first type can be called *ABComponentFactor*, which means it has effects on a particular component in a view without redirecting to other views. The other is called *ABViewFactor*. These factors have effects on the whole view by redirecting the host factor view to its variants. Despite the differences, they should both share some general properties and operations from a parent model, *ABFactor*.

From the use cases, there are a few experiments that can be implemented using the implementations of *ABComponentFactor*. These can be listed as:

ABThemeFactor: this type of factor makes it possible to test different themes on a component. The component should be an implementation of the *HasTheme* interface from Vaadin Flow[4].

ABClassFactor: the *class* factor applies different CSS classes to the target component, which must be an implementation of the *HasStyle* interface from Vaadin Flow[5].

[4]https://vaadin.com/api/com.vaadin/flow-server/1.4.2/com/vaadin/flow/component/HasTheme.html (visited on 05.03.2019)

[5]https://vaadin.com/api/com.vaadin/flow-server/1.4.2/com/vaadin/flow/component/HasStyle.html

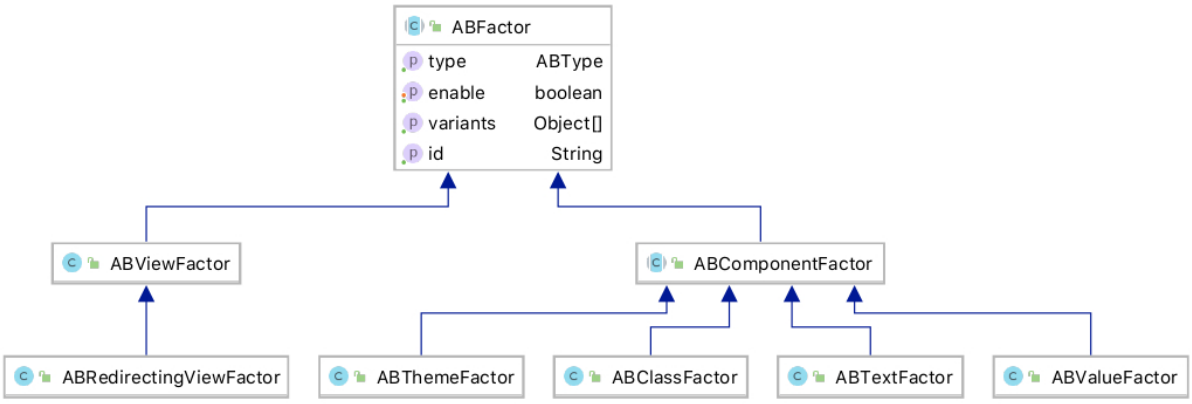

Figure 3.6: Hierarchy tree of the factor models.

ABTextFactor: text factors are applied for *HasText*[6] components to observe the results of distinct texts on a component.

ABValueFactor: value factors can be assigned to *HasValue* components. A descriptive default value could result in significant improvement for a business (section 3.1.2).

There are a few differences between these factor types but separating them into small pieces will make the code cleaner, easy to maintain and extend the existing classes to define other custom factor types.

In view type factors there are two techniques, as discussed in section 3.1: non-redirecting and redirecting views. They are implemented in this project as `ABViewFactor` and `ABRedirectingViewFactor`, respectively. These classes share most of the logic, such as validation and variant selection, but they have their distinct ways of changing to other views from a host view.

(visited on 05.03.2019)

[6]https://vaadin.com/api/com.vaadin/flow-server/1.4.2/com/vaadin/flow/component/HasText.htm (visited on 05.03.2019)l

ABFactor class

The class mainly holds shared information as a parent abstract class for the other factor types. As can be seen from figure 3.6, this class contains four properties which are described in detail as the following:

`enable`
: The property maintains the state of the factor since it should be capable of enabling or disabling during the run-time without restarting the web server.

`type`
: This type property makes it intuitive to distinguish and manage factors in the management view.

`variants`
: A simple array of objects which are candidates of the factor. Since factors must be well-defined beforehand and created once during the application initialization, since there is no demand for changing or modifying the list of variants in the middle of the application's life cycle. Hence a simple object array type is used here instead of other dominant Java Collection types.

`id` A string id of the factor which will be used as an identity when storing and fetching from a data source provider(section 3.2.4). It is also an important property for tracking purposes.

In addition, the factors observer mechanism is also implemented in this class. This is explained in greater detail in the *Observer* section (3.2.5).

ABComponentFactor and its descendant classes

`ABComponentFactor` is supposed to provide the base implementation for all kinds of factors which should apply to a `Component`[7]. The class declares two `abstract` methods which have to be implemented in the inherited classes and should only be used internally:

`void internalApply(Component component, Object variant)`

The actual logic which applies the chosen variant for the targeted component in a particular way should be implemented in this method. It has two parameters as inputs. The first one is the component which is being tested. The second parameter is the variant which has been chosen from `ABStrategy` (section 3.2.3).

`void validate(Component component)`

This method gives the flexibility for its descendants to implement their own validation rules.

In addition, `ABComponentFactor` has a `protected` method which has the default implementation choosing a variant for the factor. This method allows inherited classes to override the default behavior if necessary.

`void apply(Component component)` is the primary public API of the class that is designed for applying the current factor to a given `Component`.

`ABThemeFactor` class is an implementation of `ABComponentFactor` where the logic for testing different variants of a theme are applied to the target component. In practice, it implements the `validate` method and checks if the provided component is an instance of the `HasTheme` interface. A run-time exception will be thrown if the input is not compliant with the condition before a factor is applied in the `internalApply` method. The `HasTheme` interface has convenient APIs for adding and removing a theme, hence

[7]https://vaadin.com/api/com.vaadin/flow-server/1.4.2/com/vaadin/flow/component/package-summary.html (visited on 07.03.2019)

the chosen variant can be added to the component by calling `addThemeName` on the target component with the variant from `getVariants`.

`ABClassFactor` is the solution for developers who want to test different CSS classes on their components. The implementation details are quite similar to `ABThemeFactor` where the component is an inheritance of the `HasStyle` interface instead of `HasTheme`. This way, the tested variant will be set into the component using the `addThemeClass` method, which has to be implemented by every `HasStyle` component.

Another subclass of `ABComponentFactor` is `ABTextFactor`. The implementation is designed for diverging different texts of the component. It is only eligible for `HasText` components, which include commonly-used ones such as `Button`, `Label`, `Div`, `Headings` and others. The implementation also contains two parts: checking if the target component is an implementation of `HasText` and applying the text to the component.

The same approach is used in `ABValueFactor` to apply different values for `HasValue` components.

ABViewFactor and ABRedirectingViewFactor

`ABViewFactor` and `ABRedirectingViewFactor` are very different from `ABComponentFactor` in terms of implementation details. The reason is that manipulating the whole view is a bit trickier than modifying a single component. In particular, `ABViewFactor` has two protected methods which share common behaviors with its descendant, `ABRedirectingViewFactor`, and a public method which makes it possible to apply the current factor to a view. However, the view variants selection and transition ought to happen before actually entering the host view so that users will not get the flash rendering of the host view and then be served by the chosen view. It is likely to catch the event called `BeforeEnterEvent` thanks to the API from `BeforeEnterObserver`[8].

[8]https://vaadin.com/api/com.vaadin/flow-server/1.4.2/com/vaadin/flow/router/BeforeEnterObserver.html (visited 12.03.2019)

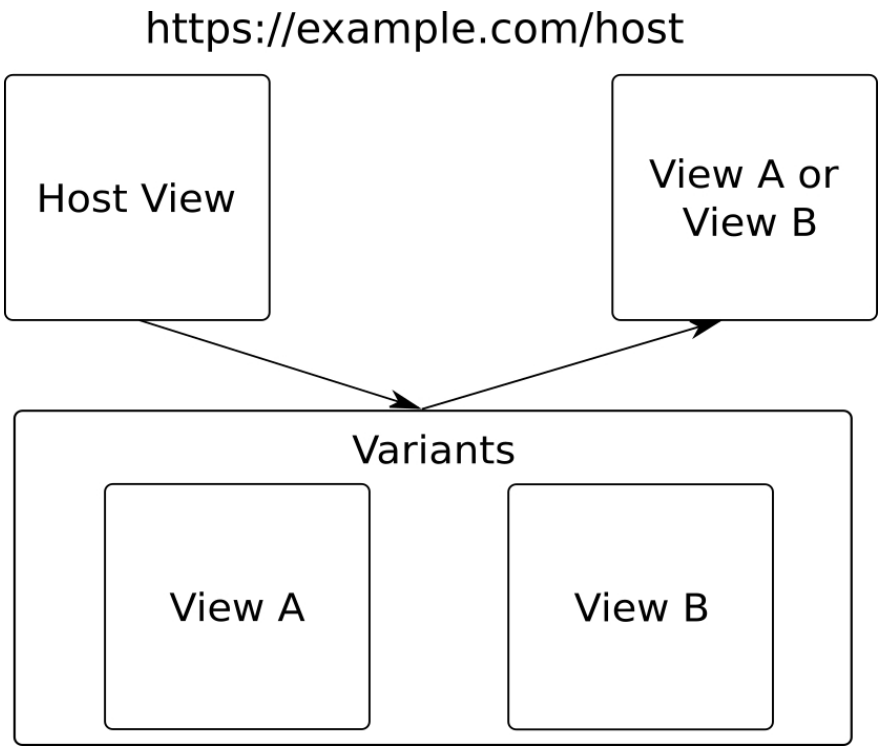

Figure 3.7: ABViewFactor's practical workflow.

By implementing the `BeforeEnterObserver` interface, the view or redirecting view factors are applicable in the method `void beforeEnter( BeforeEnterEvent event)` where the library selects the variant and navigates to them without a notable signal.

The practical workflow of a non-redirecting view factor is illustrated in figure 3.7. When a request comes to the *Host View*, it is caught in the `beforeEnter` method and evaluated by the ABHelper library. The response is eventually served either with *View A* or *View B*. Regardless of the strategy controller's decisions, the URL of the request still remains the same. For instance in figure 3.7, it is always *"https://example.com/host"*.

`ABRedirectingViewFactor` reuses the same algorithm except that after processing the `BeforeEnterEvent`, the library sends a signal to forward the *Host View* to the chosen variant. This means the URL changes to the corresponding view. As illustrated in figure 3.8, although the incoming request is *"https://example.com/host"*, after the chosen view is served, the actual URL is forwarded to the view's route, either

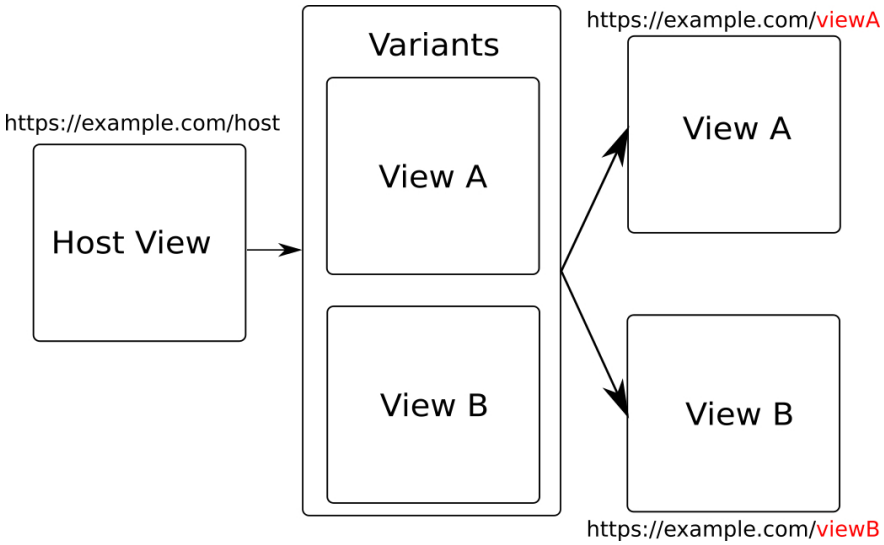

Figure 3.8: ABRedirectingViewFactor's practical workflow.

"https://example.com/viewA" or *"https://example.com/viewB"*.

Moreover, the library also considers the fact that Vaadin supports *route parameters* [9], which is a `string` following a view's route path. Thus, if there is a *Host View* request containing parameters, they are preserved and sent to the corresponding view correctly. The behavior is implemented as a part of the shared code of `ABViewFactor` and `ABRedirectingViewFactor`, so both kinds of the factor are working well with the route parameters. For example, *https://example.com/host/parameter* becomes either *https://example.com/viewA/parameter* or *https://example.com/viewB/parameter*. In this case, both *View A* and *View B* must handle the parameter by themselves; otherwise, the application will throw unexpected exceptions since the library assumes the variants are completely equivalent.

3.2.2 Factor manager

`ABManager` is a singleton which plays a middle-man role in the application. Its purpose is to ease the communication between developers, data source (section 3.2.4) and the

[9] https://vaadin.com/docs/v13/flow/routing/tutorial-router-url-parameters.html (visited 12.03.2019)

factors. Each public method of the factor manager class can be described as follows:

`public static ABManager initializeABManager(ABDataSource dataSource)`

As the coordinator connecting the actual factor model and the data source, the manager needs to know which data source it can be contacted to fetch and get the factors. This is the reason why a data source implementation is required while initializing the manager. This method should only be called once during the entire life cycle of the application. If it is called a second time, it still returns the old instance which was created in the previous initialization.

`public static ABManager getInstance()`

This method is designed for getting the initialized singleton in the entire application. If this method is executed before the `ABManager` initialization, the library throws an error with detailed messages about the problem.

`public ABDataSource getDataSource()`

This method supports for the use case that developers want to manage all created factors. It is convenient to grab the data source object and retrieve factors from there.

`public ABFactor createFactor(ABType type, String id, Object... variants)`

This method is an important API of this class. It receives three parameters containing information about the factor. At this point, the factor is checked from the data source by using the `id` field. If there is no factor with such `id` in the database, the new one with given information will be created, stored and returned. Otherwise, the data source returns the one that is found and all the information provided about the factor is discarded.

`public ABFactor createFactor(ABFactor factor)`

This method is made for the possibility of extensions. Specification of `ABType` is not required, so developers can comfortably create their factors as long as the

`ABFactor` interface is implemented. The given factor is checked in a way similar to that for the overloaded method. It is also stored to the data source if it is needed.

Since the manager class is a singleton, it should be only initialized once at the beginning of the application. In a traditional Vaadin application, it is the most straightforward method for initialization inside a `VaadinServiceInitListener`[10] implementation. The sample is shown in snippet 3.1.

Snippet 3.1: ABInitalizer example class

```
// ABInitializer.java
public class ABInitializer implements VaadinServiceInitListener {
  @Override
  public void serviceInit(ServiceInitEvent event) {
    ABMemoryDataSource dataSource = new ABMemoryDataSource();
    ABManager abManager = ABManager.initializeABManager(dataSource);
    // ... other codes
  }
}
```

3.2.3 Factor strategy controller

`ABStrategy` is a dedicated class for selecting the winning variant from a list of available options. Similar to the factor manager, `ABStrategy` is also a singleton which needs to be initialized in the server's startup phase. Hence, it provides some public static methods for the initialization and for the retrieval of the instance. The `initialize` method has an overload which accepts an integer as the expiration time for the winning variant. In other words, it is the caching time of the variant selection so that the same user receives

[10]More information about `VaadinServiceInitListener` could be found at: `https://vaadin.com/docs/v13/flow/advanced/tutorial-service-init-listener.html` (visited on 14.03.2019)

the same web interface during that period.

Randomization algorithm

The randomization algorithm is a vital element of the A/B testing system. It must satisfy three obligatory attributes[2] as below:

1. The possibility of displaying each variant is the same. There should not be any other factor which can affect the randomization decision.
2. During the testing period, a user has to receive the same decision with the assumption that the user has used the same browser without clearing the browser's cookies.
3. There should not be any relation between factors' decisions. This means that the selected variant from a factor must not have any effects on the probability of the choice for other factors.

By default, `ABStrategy` uses the Java's built-in randomization algorithm. However, developers are also capable of providing their algorithm by implementing the `ABRandomizer` interface. The interface has a `nextInt` method which receives an upper bound number and should return a valid integer from 0 to upper bound (inclusive). It is worth noting that the implementation of `ABRandomizer` has to be compliant with the third requirements by only seeding the random generator one time. This rule is satisfied by the default implementation as in snippet 3.2.

Snippet 3.2: DefaultRandomizer class

```
//DefaultRandomizer.java
public class DefaultRandomizer implements ABRandomizer {
  private final Random random = new Random();
  @Override
  public int nextInt(int upperBound) {
```

```
            return random.nextInt(upperBound);
        }
    }
```

Storing and retrieving factor variants

In order to meet the second requirement about the variants' consistency for the same user, there are two popular methods: these either save the selections in a persistent database of the server side or put them as JSON[11] objects in the browser cookies. Using the database method is expensive in terms of hardware and infrastructure setups. In this project, the data is stored in browser cookies so as to simplify the setup requirements. There is a compelling case which has been discovered in the development: because browser cookies are used on the client-side, during the first request where there is no cookie information, if two components inquire about the same factor's variant, they might produce two different results. The cause of this phenomenon is that after calculating the result for the first request, they have not been stored in the cookies yet. Subsequently, the second request comes and does not find any result in the cookies. It assumes that the factor has not been evaluated. Therefore, two requests for the same factor would have a different outcome. The problem has been resolved by saving the chosen variant in the session attributes temporarily, so that they are shared between inquiries in the same request.

3.2.4 Factor data source provider

When using the library, developers might want to store their factor data in storage. The data source provider is the middle layer between the real storage and the *ABHelper* library's APIs. There is an interface called `ABDataSource` which has defined methods for these

[11]JSON (JavaScript Object Notation) is a lightweight data-interchange format. `https://json.org/` (visited 14.03.2019)

purposes. The interface has four main functions which are: *getting a single factor by its id*, *storing a factor, deleting a factor*, and *collecting all the created factors*.

As a default implementation, `ABMemoryDataSource` has implemented the interface with an in-memory database by using a `HashMap` object. The memory data source also handles concurrency issues when adding or removing factors. It frequently runs the modifications synchronously using a lock object. An example of the getting method is shown in snippet 3.3.

Snippet 3.3: Partial implementation of ABMemoryDataSource class

```
// ABMemoryDataSource.java
// Other content
@Override
public ABFactor get(String factorId) {
  synchronized (lock) {
    return factors.get(factorId);
  }
}
// ...
```

With this approach, developers are able to choose any data source structure they prefer. It could be a persistent database for a large number of factors, or a default in-memory database for a small set of factors for prototyping. Storing data in the server's file system is also a valid choice.

3.2.5 Factor observers

Observer has been known as an effective design pattern in many programming languages. *"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."*, is an explanation from the "Design

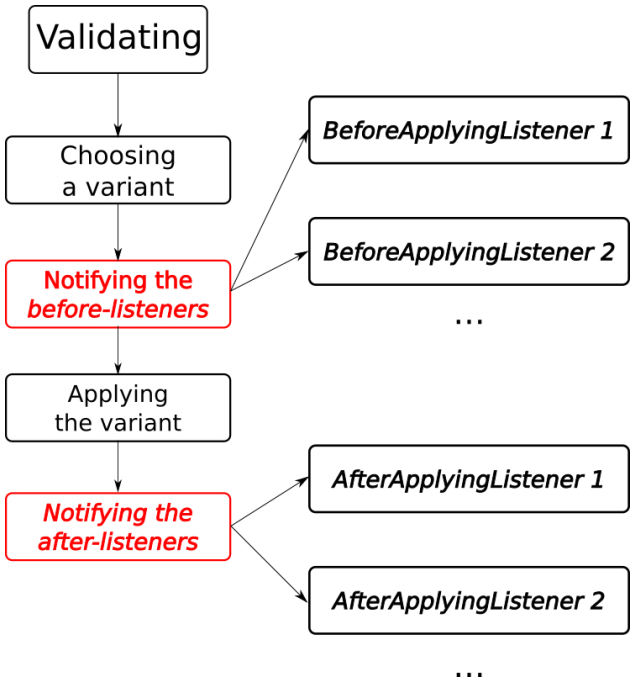

Figure 3.9: The workflow when applying a factor.

Patterns" book [11].

The Observer pattern also has been used widely with same/accurate above functionalities in the Vaadin Platform, however with a different term, `Listener`. The reason is to ensure matching with the code style of the whole Vaadin Platform implementation, where observers are widely used and referred as *"Listeners"*.

In the scope of this project, the observer pattern resolves the matter of tracking statistics. Since there is no official way to do the analytic tracking in the Vaadin Platform, developers have a freedom of choice in their use of implementations and services to track users' behaviors in their application. Accordingly, by using the observer pattern, the *ABHelper* library provides the possibility for developers to register `ABEventListener` to their factors. There are two types of the event published by a factor. One happens before applying a variant, and another is after the execution. Both of the events carry all the objective information of the factor (a component or view) and the selected variant of that factor. The simplified workflow is illustrated in figure 3.9.

3.2.6 Abstract management view

The library implements a simple abstract management view which only displays a table of factors. The view does not have any secured mechanism to protect the route, so it requires developers to extend the view and provide their access control algorithms. Otherwise, the default implemented view will be opened for anyone to view and disable (or enable) the running factors without restarting the application's server. Figure 3.10 shows the appearance of the default factor table with the information purely fetched from the `ABDataSource` (section 3.2.4).

When changing the value in checkbox columns, the checkbox will flip the `boolean` attribute of the factor to deactivate or activate itself. It is important to note that an inactive test will always return *its first variant* if requested, and all the listeners are *not* going to be notified.

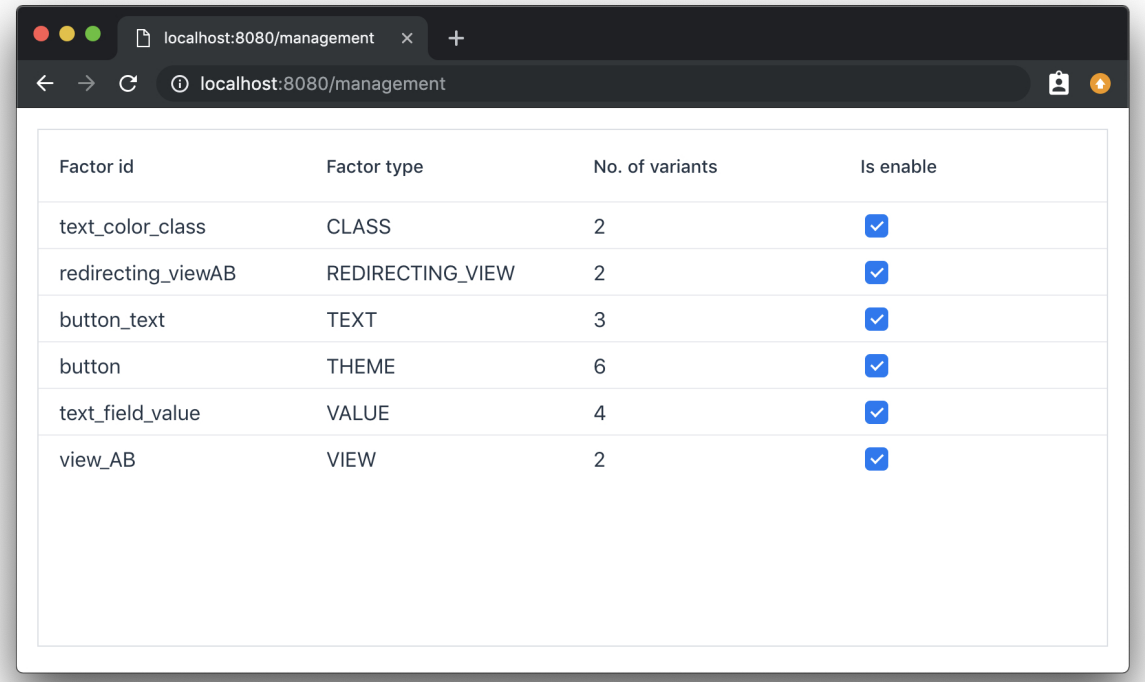

Figure 3.10: The default management view.

Chapter 4

Demonstration: A/B tests in an example application

This chapter will focus on demonstrating how the *ABHelper* library would simplify the A/B test implementation for a Vaadin application. Thus, the implemented factors in this application would be only for the purposes of library evaluation.

4.1 Example application introduction

The Vaadin Platform provides a broad set of example applications which are also called starter projects, of which the *Bookstore* project is a salient example. The `Bookstore` can be created via the Vaadin starter packs page[1] by choosing the "Simple App" option from the web page. Alternatively, it is available on GitHub[2] at https://github.com/vaadin/bookstore-starter-flow.

As shown in figure 4.1, the *Bookstore* application has three views which share a common

[1]https://vaadin.com/start (visited 18.03.2019)
[2]Software repository hosting service: https://github.com (visited 18.03.2019)

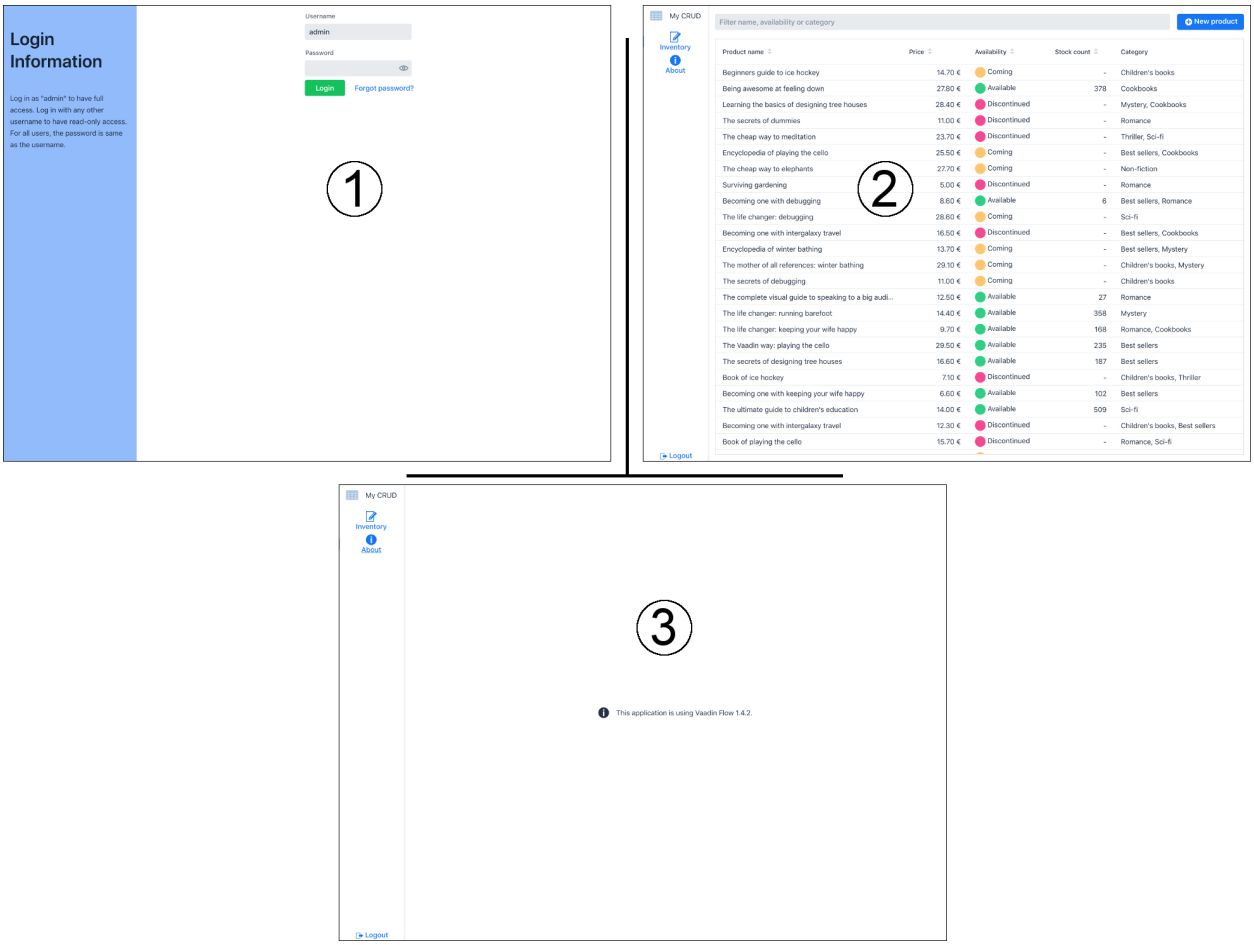

Figure 4.1: Three default views of the *Bookstore* application.

layout. In the figure, ① is *LoginView*, ② is *InventoryView*, and ③ is *AboutView*. The starter application also provides a simple authentication and access control mechanism where there is only *admin* and *user* roles. The administrator has the access right to create, update and delete any items in the *InventoryView* while others can just view it.

4.2 A/B tests implementation

4.2.1 Library's core initialization

In chapter 3, it was shown that there are several classes in the *ABHelper* library which need to be initialized before any single factor can be applied. For the project to be easier to organize, it is necessary to create a class named `BookstoreABFactors` and gather all the code related to the A/B factor there. The initialization of the library's core components should also be in that class. As can be seen from snippet 4.1, `ABInMemoryDataSource` may be used to reduce complexity for demonstration and evaluation purposes. `ABStrategy` can also be initialized with the default value of the cookie expiration time and Java's built-in random generator. The core's execution should be wrapped in a public method, as shown in an example in snippet 4.2, which also triggers other individual factor creations in later steps.

Snippet 4.1: Library's core components initialization

```
// BookstoreABFactors.java
private static void initLibraryComponents() {
  ABMemoryDataSource dataSource = new ABMemoryDataSource();
  ABManager.initializeABManager(dataSource);
  ABStrategy.initialize();
}
```

Snippet 4.2: Initialization function in BookstoreABFactors

```
// BookstoreABFactors.java
public static void initializeABFactors() {
  if (INSTANCE.isInit) {
    return;
  }
  initLibraryComponents();
  // other code
}
```

It is worth noting that for snippet 4.2, the `boolean` flag `isInitted` is needed because in the development mode of the Vaadin application, two Vaadin Servlets are running simultaneously, which calls the initialization twice. The flag thus ensures that the code will execute only once.

4.2.2 Individual component factors

A/B testing for *Theme* variant

In the *LoginView* (figure 4.1), there is a *Login* button which has two themes: *ButtonVariant.LUMO_SUCCESS* and *ButtonVariant.LUMO_PRIMARY*. This button is a good candidate for the application of a theme factor to test with other different themes.

Snippet 4.3: Create Login button factor

```
// BookstoreABFactors.java
private static void createLoginButtonFactor() {
  // Create variants
  String firstVariant = ButtonVariant.LUMO_SUCCESS.getVariantName()
      + ButtonVariant.LUMO_PRIMARY.getVariantName();
```

```
        String secondVariant = ButtonVariant.LUMO_ERROR.getVariantName()
             + ButtonVariant.LUMO_TERTIARY.getVariantName();
        String thirdVariant = ButtonVariant.LUMO_CONTRAST.getVariantName();

        // Create factor
        ABComponentFactor factor =
             (ABComponentFactor) ABManager.createFactor(
               ABType.THEME,
               LOGIN_BUTTON_THEME_FACTOR,
               firstVariant, secondVariant, thirdVariant);

        // Track factor
        factor.addAfterListener(
          abEvent -> trackComponentFactor(factor, abEvent));
    }
```

In snippet 4.3, the code creates three different variants, with the first being the default one from *Bookstore* itself and the rest being different theme variants. The factor is then created with the help of `ABManager` by specifying the factor type (`ABType.THEME`), the factor id and all the variants. The last part is about tracking the factor, which will be discussed in greater detail in section 4.2.4.

This concludes the description of theme factor initialization. In order to apply the factor on a button, more code is needed in the `LoginScreen.java`, where all the user interfaces of the *LoginView* are defined. Instead of explicitly adding themes to the login button, `ABController` can be used to apply a factor on it. During the application run time, ABHelper library will select one theme from the declared variants of the tests for the login button. The modified code for creating the button is shown in snippet 4.4.

Snippet 4.4: Create Login button and apply theme factor on it

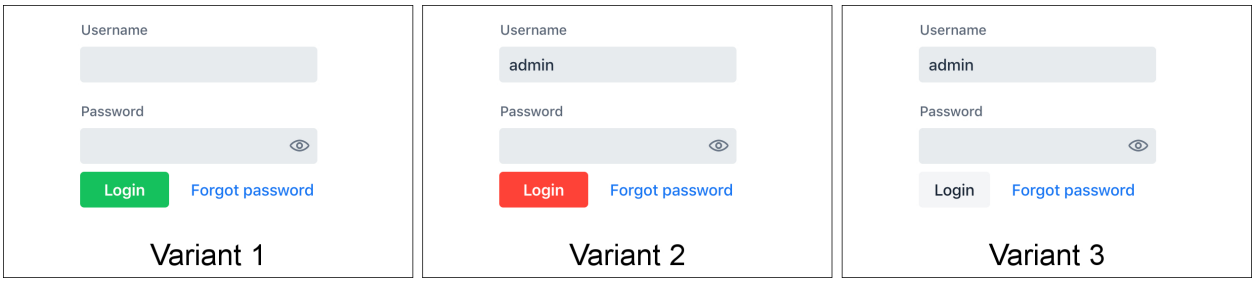

Figure 4.2: Login button for different variants of an A/B theme test.

```
login = new Button("Login");
login.setId("loginButton");
ABController.applyFactor(login,
        BookstoreABFactors.LOGIN_BUTTON_THEME_FACTOR_ID);
```

As a result, the login page will randomly load three different variants for the login button for random users. The difference between the three variants is shown in figure 4.2.

Other types of component factors

Other types of factors (text, value and CSS class) can be implemented using the same steps:

- Create a factor in `BookstoreABFactors.java`.
- Find the target component which the created factor should be applied on.
- Find the code where that component explicitly sets the value which should vary between different variants of the factor. Then replace that code by applying the factor using the `ABManager.applyExperiment` method.

Figure 4.3 shows an example of testing different texts for the *Forgot password* button by following the steps above.

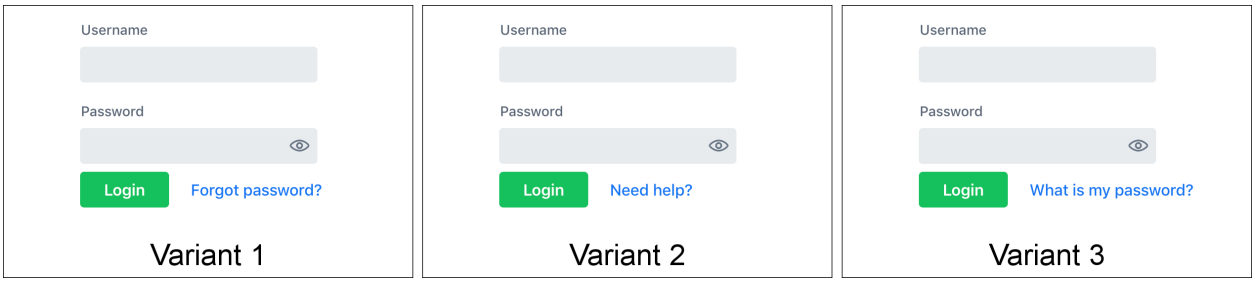

Figure 4.3: "Forgot Password" button with different text content.

4.2.3 View factors

Non-redirecting view factors

Non-redirecting view factors are usually used to test distinctive back-end algorithms in a view. Within the *Bookstore* application, *InventoryView* can execute different back-end functions in the same view. Assume that there is a demand to test the `editProduct` function in *InventoryView*. The alternative view should show a small notification before editing a product.

First of all, the original view should change the route path to something different from its default one, for instance, `Inventory_A`. At this point, the variant view should be created and extend from the original view. It should also have a different route which can be called `Inventory_B`. Those routes can be accessed normally as with other views, but during the experiment, users will not notice any change in the URL since the `AB helper` library will always show the original URL instead of each variant view's URL. The source code for the treatment view could be as in snippet 4.5.

Snippet 4.5: Source code of the alternative view of *InventoryView*

```
// AlternativeSampleCrudView.java
@Route(value = "Inventory\_B", layout = MainLayout.class)
public class AlternativeSampleCrudView extends SampleCrudView {
  public AlternativeSampleCrudView() {
    super();
  }

  @Override
  public void editProduct(Product product) {
    super.editProduct(product);
    if (product != null) {
      String text = String.format("You are about to edit the product: '%s'",
      product.getProductName());
      Notification.show(text);
    }
  }
}
```

For the next step, an intermediate view is needed for receiving the request from original *InventoryView*. Hence, it should have the same route configuration as the original one.

Snippet 4.6: The intermediate view between two view variants.

```
@Route(value = "Inventory", layout = MainLayout.class)
@RouteAlias(value = "", layout = MainLayout.class)
public class ABInventoryView extends VerticalLayout
implements BeforeEnterObserver, HasUrlParameter<String> {

  @Override
  public void beforeEnter(BeforeEnterEvent event) {
```

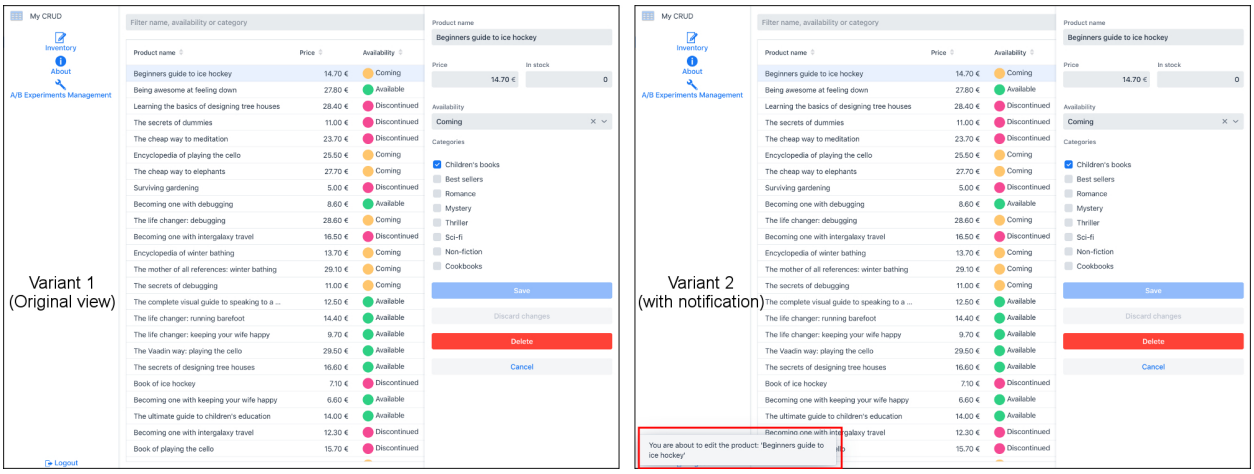

Figure 4.4: Comparison of non-redirecting views.

```
    ABController.applyViewFactor(event,
        BookstoreABFactors.INVENTORY_VIEW_FACTOR);
    }

    @Override
    public void setParameter(BeforeEvent event,
    @OptionalParameter String parameter) {
        // no operation, the parameter will be forwarded automatically by
        // the ABHelper library
    }
}
```

As is apparent from snippet 4.6, `ABInventoryView` has implemented `BeforeEnterObserver` to apply the view factor in the `beforeEnter` event. This class has also implemented the `HasUrlParameter<String>` interface because the original *InventoryView* accepts the `String` parameter in the route. By just implementing the interface without any code inside `setParameter`, the view will be able to receive the parameter when applying the factor. This helps the ABHelper library forward given parameters to the selected view.

Finally, what the two different view variants will look like is shown in figure 4.4. The second variant indicates a small notification when a user wants to edit a product. Obviously, the logic in the second view is just as an illustration of how the work could be done. The alternative view could test any different back-end algorithms as required.

Redirecting view factor

Unlike the non-redirecting view factor, it is expected that the URL of a web page will change in accordance with the selected variant. However, they both share the steps for setting up the factor. In the *Bookstore* application, *AboutView* can be used as an example for this factor. Assume that there is a requirement for completely new *AboutView* page called *NewAboutView*. The current *AboutView* would be changed to *OldAboutView*. The intermediate view for receiving requests will become *AboutView*.

Snippet 4.7: The intermediate view of an A/B test for *AboutView*

```
@Route(value = "About", layout = MainLayout.class)
@PageTitle("About")
public class AboutView extends HorizontalLayout implements BeforeEnterObserver {
  // Other code...
  @Override
  public void beforeEnter(BeforeEnterEvent event) {
    ABController.applyViewFactor(event,
      BookstoreABFactors.ABOUT\_VIEW\_FACTOR);
  }
}
```

As it is shown in snippet 4.7, the redirecting view factor also happens inside the `beforeEnter` event. It means that when the request comes with the route `/About`, it will be forwarded to the either *NewAboutView* or *OldAboutView*. Figure 4.5 shows the comparison of *NewAboutView*

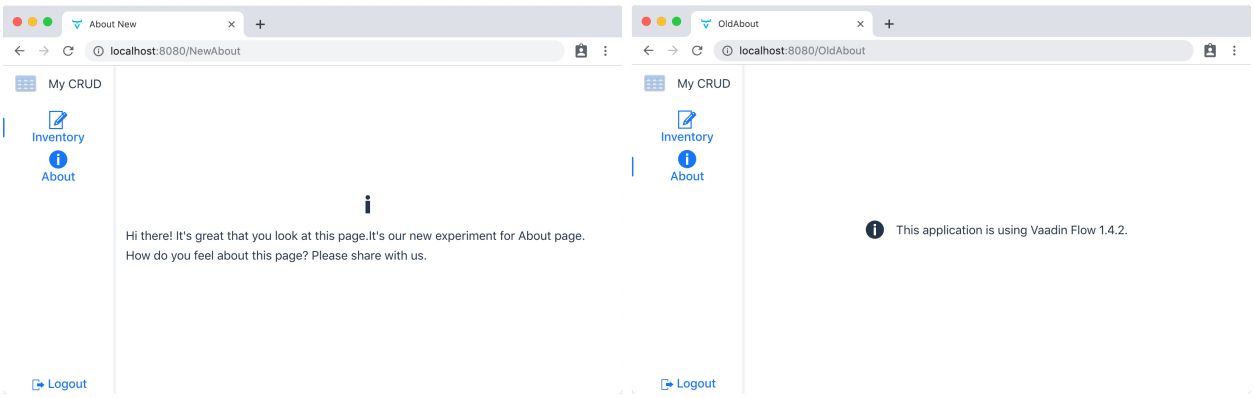

Figure 4.5: *NewAboutView* and *OldAboutView* comparison.

and *OldAboutView*. The differences are text content and alignment of the icon. It is made intentionally in order to simplify the example. Technically, two views can be implemented in a completely different way and accessed separately using their URLs.

4.2.4 Tracking A/B factors

As it is mentioned in section 3.2.5, there is no official way of analytic tracking for a Vaadin application. Developers can choose any technique depending on their requirements. However, there is a community project which is called *Vaadin Google Analytics Tracker*[3]. To demonstrate how the analytic tracking works in combination with ABHelper, the *Vaadin Google Analytics Tracker* is sufficient since it is simple to use and does the job.

With assistance from the *Vaadin Google Analytics Tracker*, two methods for tracking purposes can be created in the `BookstoreABFactors`. One is for tracking component tests while another is used for the view tests. The implementations are shown in snippet 4.8 and 4.9. While tracking views, it is necessary to fetch the URL of the view from `Route` annotation of the selected variant. On the other hand, in component A/B tests, the variant is usually a string, so it is likely to get them by calling the `toString` method. Also noteworthy is the fact that these examples merely show how data tracking can be done by showing the

[3]The project is available at `https://github.com/samie/vaadin-ga-tracker`.

simplest working implementation. The collected data might not be useful for use cases in practice because this study does not focus on what data should be collected for analysis.

After defining the tracking methods, they must be added as an `afterListener` in every factor to be tracked. Thus, after applying a variant in a test, the tracking method will be executed to send data for analytic tracking. In figure 4.6, it is indicated that four events have been tracked. Each event action corresponds to a factor id in the application. The event label is the value of the selected variant in that test. The tracking data can be anything; it is entirely up to application developers.

Snippet 4.8: Implementation of analytic tracking for Components

```
private static void trackComponentFactor(ABComponentFactor factor,
                                         ABEvent abEvent) {
  GoogleAnalyticsTracker.getCurrent().sendEvent("AB_COMPONENT_TEST",
    factor.getId(), abEvent.getSelectedVariant().toString());
}
```

Snippet 4.9: Implementation of analytic tracking for Views

```
private static void trackViewFactor(ABViewFactor factor, ABEvent abEvent) {
  String trackingAction = factor.getId();
  String value = getViewVariantValue(abEvent);
  GoogleAnalyticsTracker.getCurrent().sendEvent("AB_VIEW_TEST",
          trackingAction, value);
}
```

4.2.5 A/B factors management view

The *ABHelper* library provides an abstract class for managing the A/B tests in an application. Developers could extend the view and provide their protection mechanism. In the *Bookstore*

	Event Action	Event Label
1.	forgot_password_text_test	Need help?
2.	information_class_test	color-yellow
3.	login_button_test	contrast
4.	username_value_test	manager

Figure 4.6: A/B tests are captured in Google Analytics.

example, it should be possible to display the view only if the username is *"admin"*. If it does not meet the given condition, the view should be *rerouted* to an *ErrorView*. The checking must be implemented in a `beforeEnter` event. The whole management view implementation is described in snippet 4.10.

Snippet 4.10: Implementation of analytic tracking for Views

```
@Route(value = "ABManagement", layout = MainLayout.class)
@IgnorePageView
public class BookstoreABManagementView extends ABAbstractManagementView
                                        implements BeforeEnterObserver {
  public static final String VIEW_NAME = "A/B Factors Management";

  @Override
  public void beforeEnter(BeforeEnterEvent event) {
    if (!AccessControlFactory.getInstance().createAccessControl()
                .isUserInRole(AccessControl.ADMIN_ROLE_NAME)) {
      event.rerouteTo(ErrorView.class);
    }
  }
}
```

4.3 Creating custom factors

As a consequence of being developed as a library, `ABHelper` will never satisfy all the needs of web application developers. They sometimes might wish to have their customized type of factors that meet their requirements. By keeping that use case during the development phase of `ABHelper`, it is possible to create custom factors without much effort.

Assume that in the *Bookstore* application the developer wants to test different click listeners for the *Forgot Password* button in *LoginView*. The requirement can be met by using a view factor. However, it would be a little simpler, if there were a new type of factor called *ListenerFactor* which can randomly get one listener from two different implementations.

In order to add a new factor type, developers can create a new class extending the `ABComponentFactor` or `ABViewFactor`. In this case, the `ListenerFactor` will inherit from the `ABComponentFactor` because its target is in component type. The implementation will need additional information about `EventType` so it should be passed on as a parameter in the constructor. There are two methods which must be implemented in the `ListenerFactor`:

`validating` is for certifying before the application of the factor. In this case, it needs to verify that all the submitted variants are in the type of `ComponentEventListener`. By violating this condition, an illegal state will be thrown to stop the application.

`internalApply` is to apply the selected variant to the component. The implementation uses `ComponentUtil` from Vaadin Flow[4] to add the selected listener to the component with the provided event type.

The source code of the custom factor type is shown in snippet 4.11. After this point, the *Bookstore* application has had a new factor type which can be applied to any components.

Snippet 4.11: Implementation of ListenerFactor

[4]Vaadin Flow is the core framework of Vaadin Platform. More information can be found in `https://vaadin.com/docs/flow/Overview.html` (visited on 24.03.2019)

```
public class ListenerFactor extends ABComponentFactor {
  private final Class<? extends ComponentEvent> eventType;

  public ListenerFactor(String id,
      Class<? extends ComponentEvent> eventType, Object... ab) {
    super(ABType.CUSTOM, id, ab);
    this.eventType = eventType;
  }

 @Override
  protected void internalApply(Component component, Object selectedVariant) {
    ComponentUtil.addListener(component, eventType,
                  (ComponentEventListener) selectedVariant);
  }

  @Override
  protected void validate(Component component) {
    for (Object variant : getVariants()) {
      if (!(variant instanceof ComponentEventListener)) {
        throw new IllegalStateException(
          "Variants must implement 'ComponentEventListener' class");
      }
    }
  }
}
```

In order to test different listeners, a new listener factor should be created first. The initialization can be made similarly to other factors in `BookstoreABFactors`. Snippet 4.12 shows the creation of a click listener test with two different implementations. One

presents a notification in the bottom left of the view while the other will display the notification in the middle of the screen.

Snippet 4.12: Implementation of ListenerFactor

```
public class ListenerFactor extends ABComponentFactor {
  private final Class<? extends ComponentEvent> eventType;

  public ListenerFactor(String id,
  Class<? extends ComponentEvent> eventType, Object... ab) {
    super(ABType.CUSTOM, id, ab);
    this.eventType = eventType;
  }

  @Override
  protected void internalApply(Component component, Object selectedVariant) {
    ComponentUtil.addListener(component, eventType,
      (ComponentEventListener) selectedVariant);
  }

  @Override
  protected void validate(Component component) {
    for (Object variant : getVariants()) {
      if (!(variant instanceof ComponentEventListener)) {
          throw new IllegalStateException(
            "Variants must implement 'ComponentEventListener' class");
      }
    }
  }
}
```

Snippet 4.13: Creation of ClickListener factor

```
private static void createClickListenerFactor() {
  ComponentEventListener<ClickEvent<Button>> defaultListener =
    event -> Notification.show("I am the default listener");
  ComponentEventListener<ClickEvent<Button>> alternativeListener =
  event -> Notification.show("I am the alternative listener", 5000,
       Notification.Position.MIDDLE);
  ABComponentFactor abComponentFactor = new ListenerFactor(
    LISTENER_FACTOR, ClickEvent.class, defaultListener, alternativeListener);
  ABManager.createFactor(abComponentFactor);
}
```

In this example, the factor will be applied to the *Forgot Password* button in the *LoginView* with one line of code `ABController.applyFactor(forgotPassword, BookstoreABFactors.LISTENER_FACTOR);`.

4.4 Running an example experiment

4.4.1 Defining an example experiment

In the *BookStore* application, assume that only 10% of customers actually buy a book while being at the *Inventory* view. A hypothesis is established that customers do not make a purchase because the *Buy* button is not attractive. Subsequently, the *BookStore* owner wants to conduct an experiment to verify the validity of this hypothesis. In this case, the button's theme is a factor. The owner decides to add a new theme variant for the button to make it more attractive. The default variant (control) is *primary* and the new variant (treatment) is *success primary*. Figure 4.7 illustrates the two themes of the *Buy* button. ① is the default value (`primary`) and ② is the new value (`success primary`). In setting up

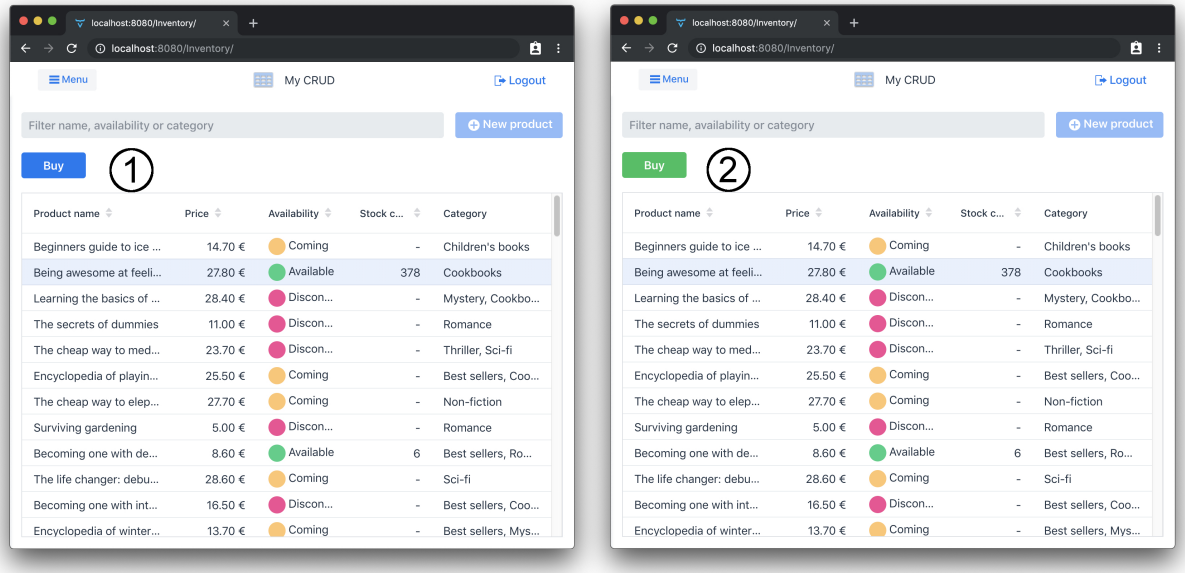

Figure 4.7: Different variants of *Buy* button in BookStore.

the experiment, the owner wants to have a 5% increment in the number of customers who place orders in the store. This is the overall evaluation criterion (OEC) of the experiment.

4.4.2 Implementing the experiment

According to the experiment defined in section 4.4.1, a new theme factor must be implemented in the *BookStore*. Similar to other factors, it needs to be created in `BookstoreABFactors.java` with an id, for instance: *buy_button_theme_factor*. Snippet 4.14 shows the method used to create the factor with given details. It also adds a listener to send *Google Analytics* data for further analysis.

Snippet 4.14: Creation of Buy button factor

```
private static void createBuyButtonThemeFactor() {
  // Create variants
  String control = ButtonVariant.LUMO_PRIMARY.getVariantName();
```

```
        String treatment = ButtonVariant.LUMO_SUCCESS.getVariantName() + " " +
         ButtonVariant.LUMO_PRIMARY.getVariantName();
        // Create factor
        ABFactor factor = ABManager.createFactor(ABType.THEME,
        BUY_BUTTON_THEME_FACTOR, control, treatment);
        factor.addAfterListener(abEvent -> {
           GoogleAnalyticsTracker.getCurrent().sendEvent("AB",
           BUY_BUTTON_THEME_FACTOR,
           abEvent.getSelectedVariant().toString());
         });
    }
```

Afterwards, the newly-created factor should be applied to the *Buy* button. In addition, when customers purchase a book, it is also necessary to track which variant is involved in the session. In order to accomplish this, the button has to send another event to *Google Analytics*. The event should express the *experiment's id*, *factor's id* and its value. The details are needed because there might be more factors associated with an experiment, although in this case there is only one factor. Snippet 4.15 demonstrates the process of applying the factor to the `Buy` button.

Snippet 4.15: Appling the theme factor to the Buy button

```
// other code ...
Button buyButton = new Button("Buy");
buyButton.setEnabled(false);
ABController.applyFactor(buyButton,
BookstoreABFactors.BUY_BUTTON_THEME_FACTOR);
buyButton.addClickListener(buttonClickEvent -> {
   // other codes...
   GoogleAnalyticsTracker.getCurrent().sendEvent(
   BUY_RATE_EXPERIMENT,
```

```
        BookstoreABFactors.BUY_BUTTON_THEME_FACTOR,
        buyButton.getThemeName());
    });
```

4.4.3 Analyzing the experiment's result

According to formula 2.1, the standard deviation of the sample data is needed for calculating the number of users who should be involved in the experiment. The purchasing rate can be modeled as a Bernoulli trial with *p = 0.1* (10%), which is the probability of buying books prior to the experiment. Thus, the standard deviation is calculated with the equation $\sqrt{p \times (1-p)}$ [2, page 153]. The difference to be detected, $\triangle$, is $10\% \times 5\% = 0.005$. Therefore, the number of users for this experiment should be 57,600 according to formula 4.1. It is noteworthy that the number is calculated with 95% in confidence level and 80% power.

$$16 \times \frac{\sqrt{0.1 \times (1-0.1)}}{(0.1 \times 0.05)^2} = 57,600. \tag{4.1}$$

The period for running the experiment might vary depending on the application traffic. When the data size is fulfilled, it can be evaluated using formula 2.2, assuming that the experiment has been executed for a sufficient period to collect data from at least the minimum sample size. The distribution is described in the following table:

Variants/Data	No. of users	Conversion rate (OEC)
Variant A	30,000	10.0%
Variant B	30,000	12.3%

From the collected data, the estimated deviation of the difference between two conversion

rates can be calculated as:

$$\widehat{\sigma_d} = \sqrt{\frac{p_A \times (100 - p_A)}{n_A} + \frac{p_B \times (100 - p_B)}{n_B}} = \sqrt{\frac{10 \times (100 - 10)}{30,000} + \frac{12.3 \times (100 - 12.3)}{30,000}} = 0.257 \tag{4.2}$$

The result of the *t-test* in formula 2.2 is $t = \frac{\overline{O_B} - \overline{O_A}}{\widehat{\sigma_d}} = \frac{12.3 - 10.0}{0.257} = 8.895$. Evidently, the value is greater than the pre-defined threshold *1.96*, so the null hypothesis H_0 has to be rejected. The result indicates that the change in treatment (variant B) brings a positive improvement and should replace the control (variant A).

4.5 Summary

In this chapter, the *Bookstore* application has been modified to implement A/B tests by using the `ABHelper` library. The implementation requires a few initializing steps, but to add more tests, there just needs to be the creation of factors and application of these to the target components or views. The source code of the example project is available at `https://github.com/qtdzz/bookstore-starter-flow/tree/abtests`. Moreover, the library can be used in all other Vaadin applications in the same way. It also provides the basic structure and APIs for creating custom test cases which fit with specific requirements.

Chapter 5

Advantages and limitations of the project

The implementation of the example project in chapter 4 can be considered as an experiment of using the library for implementing A/B tests in a Vaadin application. During the period of working with the example, the `ABHelper` library has shown that it is convenient and offers significant improvements compared to implementing A/B tests from scratch. In addition, there are a few limitations that should be corrected and improved in later versions. Those strengths and drawbacks are summarized in this chapter.

5.1 Advantages

5.1.1 Simplicity

The primary purpose and motivation of *ABHelper* is to simplify the implementation of A/B tests in Vaadin applications. It has been demonstrated in the example application that after creating a factor with a minimal amount of code, the factor can be applied to a component by adding one line of modification. In addition, the random generator and data source have their default implementations which are suitable for most of the

cases and easy to customize for different requirements. The library also provides APIs for injecting observers to the factor for obtaining all the information about the A/B tests. The convenient APIs help developers to handle the analytic tracking with a simple and straightforward approach. A predefined management view comes with the library to help manage created factors efficiently without restarting the web server. It provides a convenient way to stop doing a test by falling to the default variant.

5.1.2 Reusability

As the nature of a library, *ABHelper* is an independent module which can be used in any Vaadin application. It assists in reducing duplicated code, in the event that a development team needs to implement A/B tests in multiple applications. Being a separate module, it allows the library to grow independently where it can be tested thoroughly without any hassles with the host application.

5.1.3 Flexibility for extending

The library provides ways for extending and customizing as much as possible. Application developers could use their algorithms to generate random selections for choosing variants. They can use their implementation of the data source to store and fetch the factors. It is also possible to create custom factor types for their specific requirements without writing a lot of additional code. If the use case is general and popular enough, it could be contributed back to the library which will improve the library itself. Consequently, the application would be able to have a custom test without any significant customization.

5.2 Limitations

5.2.1 Developer experience

The project has been developed as an independent module which does not interfere deeply in the internal workflow of the Vaadin Platform, and its purpose is to modify application behaviors so that it requires some initializing codes which is slightly more than a usual Vaadin add-on in the case of view factors. The exposed APIs of the library are not optimal because they have not been used in any other projects than the example application. Once receiving external feedback, the situation could be improved by adjusting and fine-tuning the APIs to be more friendly and easy to use.

5.2.2 Lacking of diversity in factors types

In the scope of this research, *ABHelper* only provides seven types of factors which might be insufficient in a real application. It manages to create a generic type which could match more use cases, but that is not the chosen way. The library implements a factor type for a specific use case rather than combining them as a general case. In this approach, the library can do better in the validation phase where it can reduce the possibility of `RuntimeException`, but the initial diversity of the types has been sacrificed for that.

Chapter 6

Conclusion and future work

6.1 Conclusion

In this thesis, the *ABHelper* library has been implemented as a solution for simplifying the implementation of A/B tests in a Vaadin application. The library is implemented as an independent module which provides flexible APIs and a structure for facilitating the performance of A/B tests.

The library extracts the core components and functions of A/B tests from the host application, so it facilitates developers who use *ABHelper* in their application to simplify the process of creating and managing A/B tests. They can apply factors not only for different UI components but also for various views. `ABComponentFactor` makes it possible to test different variants on Vaadin's components by just creating a factor and applying it with less than ten lines of code (for example in snippet 4.13). For view factors, more code is required to create an intermediate view (snippet 4.7) and configure the two variant views. However, all the work could be done in Java code of the application without traffic splitting or page rewriting on a web server, as with traditional methods [2].

By using the *ABHelper* library, all the data from factors can be collected using any tools or

platform. The tracking feature could be done by attaching observers for the target factors. In the example project in chapter 4, a community add-on has been used to send the factor data to Google Analytics with very minimal additional code (snippet 4.8 and 4.9).

Moreover, application developers could benefit from the predefined management view providing a simple table which contains all the information about factors implemented in the application. The management view makes it possible to enable or disable a factor during the application run-time without restarting the web server. The component which is in charge of the disabled factor will fall back to the first variant defined in the factor.

6.2 Future work

For the future development of the project, there are many aspects which could be optimized and improved, especially, the points which are mentioned in the limitation section (5.2). In the event that the library could gather feedback from developers, the APIs could be changed and optimized for simplifying the process. There seems to be high demand for more factor types to fit with numerous types of applications, since they always have their particular requirements.

Moreover, the management view is the area with the greatest potential for improvement. Some nice features could be added to it, such as making it possible to alter the factor's variants during the run-time. This feature would allow people who are not developers (for example employees from the marketing team) to modify the factor as they desire.

In conclusion, it would be beneficial if the library could provide a built-in analytic system which collects and shows factors' data in a view. The analytic could leverage other components from the Vaadin Platform to make the look more useful and attractive.

Bibliography

[1] *Stack Overflow Developer Survey 2018*. Stack Overflow. URL: https://stackoverflow.com/insights/survey/2018/ (visited on 03/25/2019).

[2] Ron Kohavi et al. "Controlled experiments on the web: survey and practical guide". In: *Data Mining and Knowledge Discovery* 18.1 (Feb. 2009), pp. 140–181. ISSN: 1573-756X. DOI: 10.1007/s10618-008-0114-1. URL: https://doi.org/10.1007/s10618-008-0114-1.

[3] Pawan Vora. *Web Application Design Patterns*. Google-Books-ID: S2kv6RJztQcC. Morgan Kaufmann, Mar. 12, 2009. 469 pp. ISBN: 978-0-08-092145-7.

[4] *World Internet Users Statistics and 2018 World Population Stats*. URL: https://www.internetworldstats.com/stats.htm (visited on 02/19/2019).

[5] Harry McCracken. *How Gmail Happened: The Inside Story of Its Launch 10 Years Ago*. Time. URL: http://time.com/43263/gmail-10th-anniversary/ (visited on 02/19/2019).

[6] Team EffectiveUI et al. *Effective UI*. ISBN: 978-0-596-15478-3. URL: http://shop.oreilly.com/product/9780596154790.do (visited on 02/22/2019).

[7] Greg Linden. *Geeking with Greg: Early Amazon: Shopping cart recommendations*. Geeking with Greg. Apr. 25, 2006. URL: https://glinden.blogspot.com/2006/04/early-amazon-shopping-cart.html (visited on 03/26/2019).

[8] Ranjit K. Roy. *Design of Experiments Using The Taguchi Approach 16 Steps to Product and Process Improvement*. Wiley-Interscience, 2001. ISBN: 978-0-471-36101-5 0-471-36101-1.

[9] *Optimizely: The Big Book of Experimentation Case Studies*. URL: https://www.optimizely.com/resources/experimentation-case-studies/ (visited on 01/31/2019).

[10] Dan Siroker, Pete Koomen, and Cara Harshman, eds. *A/B Testing: The Most Powerful Way to Turn Clicks into Customers*. Hoboken, NJ, USA: John Wiley & Sons, Inc., Jan. 2, 2012. ISBN: 978-1-119-17645-9 978-1-118-53609-4. DOI: 10.1002/9781119176459. URL: http://doi.wiley.com/10.1002/9781119176459 (visited on 01/31/2019).

[11] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 edition. Reading, Mass: Addison-Wesley Professional, Nov. 10, 1994. 395 pp. ISBN: 978-0-201-63361-0.